Generics

Summary

- Generics allow us to create reusable classes, interfaces and functions.
- A generic type has one or more generic type parameters specified in angle brackets.
- When using generic types, we should supply arguments for generic type parameters or let the compiler infer them (if possible).
- We can constrain generic type arguments by using the **extends** keyword after generic type parameters.
- When extending generic classes, we have three options: can pass on generic type parameters, so the derived classes will have the same generic type parameters. Alternatively, we can restrict or fix them.
- The **keyof** operator produces a union of the keys of the given object.
- Using type mapping we can create new types based off of existing types. For example, we can create a new type with all the properties of another type where these properties are readonly, optional, etc.
- TypeScript comes with several utility types that perform type mapping for us. Examples are: **Partial<T>**, **Required<T>**, **Readonly<T>**, etc.
- See the complete list of utility types:

  https://www.typescriptlang.org/docs/handbook/utility-types.html

Cheat Sheet

Generic classes

```
class KeyValuePair<K, V> {
  constructor(public key: K, public value: V) {}
}

let pair = new KeyValuePair<number, string>(1, 'a');

// The TypeScript compiler can sometimes infer
// generic type arguments so we don't need to specify them.
let other = new KeyValuePair(1, 'a');
```

Generic functions

```
function wrapInArray<T>(value: T) {
  return [value];
}

let numbers = wrapInArray(1);
```

Generic interfaces

```
interface Result<T> {
  data: T | null;
}
```

Generic constraints

```
function echo<T extends number | string>(value: T) {}

// Restrict using a shape object
function echo<T extends { name: string }>(value: T) {}

// Restrict using an interface or a class
function echo<T extends Person>(value: T) {}
```

Extending generic classes

```
// Passing on generic type parameters
class CompressibleStore<T> extends Store<T> { }

// Constraining generic type parameters
class SearchableStore<T extends { name: string }> extends Store<T> { }

// Fixing generic type parameters
class ProductStore extends Store<Product> { }
```

The keyof operator

```
interface Product {
  name: string;
  price: number;
}

let property: keyof Product;
// Same as
let property: 'name' | 'price';

property = 'name';
property = 'price';
property = 'otherValue'; // Invalid
```


Type mapping

```ts
type ReadOnly<T> = {
  readonly [K in keyof T]: T[K];
};

type Optional<T> = {
  [K in keyof T]?: T[K];
};

type Nullable<T> = {
  [K in keyof T]: T[K] | null;
};
```

Utility types

```ts
interface Product {
  id: number;
  name: string;
  price: number;
}

// A Product where all properties are optional
let product: Partial<Product>;

// A Product where all properties are required
let product: Required<Product>;

// A Product where all properties are read-only
let product: Readonly<Product>;

// A Product with two properties only (id and price)
let product: Pick<Product, 'id' | 'price'>;

// A Product without a name
let product: Omit<Product, 'name'>;
```

